Collective Shortest Paths for Minimizing Congestion on Temporal Load-Aware Road Networks

Chris Conlan
University of Warwick
Coventry, United Kingdom
chris.conlan@warwick.ac.uk

Teddy Cunningham
University of Warwick
Coventry, United Kingdom
teddy.cunningham@warwick.ac.uk

Gunduz Vehbi Demirci
University of Warwick
Coventry, United Kingdom
gunduz.demirci@warwick.ac.uk

Hakan Ferhatosmanoglu
University of Warwick
Coventry, United Kingdom
hakan.f@warwick.ac.uk

ABSTRACT

Shortest path queries over graphs are usually considered as isolated tasks, where the goal is to return the shortest path for each individual query. In practice, however, such queries are typically part of a system (e.g., a road network) and their execution dynamically affects other queries and network parameters, such as the loads on edges, which in turn affects the shortest paths. We study the problem of collectively processing shortest path queries, where the objective is to optimize a collective objective, such as minimizing the overall cost. We define a temporal load-aware network that dynamically tracks expected loads while satisfying the desirable 'first in, first out' property. We develop temporal load-aware extensions of widely used shortest path algorithms, and a scalable collective routing solution that seeks to reduce system-wide congestion through dynamic path reassignment. Experiments illustrate that our collective approach to this NP-hard problem achieves improvements in a variety of performance measures, such as, i) reducing average travel times by up to 63%, ii) producing fairer suggestions across queries, and iii) distributing load across up to 97% of a city's road network capacity. The proposed approach is generalizable, which allows it to be adapted for other concurrent query processing tasks over networks.

CCS CONCEPTS

• **Information systems** → **Spatial-temporal systems**; *Graph-based database models*; *Location based services*; Query optimization; • **Theory of computation** → *Shortest paths.*

KEYWORDS

Shortest Path Queries, Road Networks, Temporal Load-aware Networks, Collective Route Optimization, Graph Data Analytics

IWCTS'21, November 1, 2021, Beijing, China

ACM ISBN 978-1-4503-9117-7/21/11…$15.00
https://doi.org/10.1145/3486629.3490691

ACM Reference Format:
Chris Conlan, Teddy Cunningham, Gunduz Vehbi Demirci, and Hakan Ferhatosmanoglu. 2021. Collective Shortest Paths for Minimizing Congestion on Temporal Load-Aware Road Networks. In *14th ACM SIGSPATIAL International Workshop on Computational Transportation Science (IWCTS'21), November 1, 2021, Beijing, China.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3486629.3490691

1 INTRODUCTION

The shortest path query over networks is a widely studied problem with well-known and efficient solutions. The most common setting is to consider each query in isolation, where the objective is to optimize each individual path. However, in practice, multiple queries are executed simultaneously and each query influences the cost incurred by other queries. Navigational services are a real-world example of executing concurrent shortest path queries. Simply targeting the fastest route[1] for each query independently can induce congestion as the recommended routes are likely to share common edges. Consequently, one must aim for an aggregate optimization goal that seeks alternative routes for some vehicles in order to reduce system-wide congestion and decrease total travel times. To formally address the problem, in this paper, we introduce a temporal load-aware setting in which the network tracks expected future load in the system based on the queries' origin and destinations. An arrival time function uses the load to penalize congestion, which is integrated into the shortest path algorithms so that congestion is proactively avoided. Our goal is to address the problem in a collective manner (i.e., we seek optimal routes for a set of queries concurrently), rather than calculating routes independently for chronologically encountered queries.

Figure 1 is a simple illustration of the principle of collectively processing shortest path queries, and demonstrates how anticipating future congestion can provide alternative paths that are globally optimal. In this example, there is a demand to route six vehicles from node 1 to node 4 and each edge has capacity for four cars to travel at maximum speed. If loads exceed capacity, we assume all vehicles experience delays. A traditional navigational service may route all six vehicles along path 1-2-3-4, which would cause congestion and delays along these edges. A collective solution would direct two vehicles along route 1-5-3-4, two vehicles along 1-2-3-4,

[1]We use the terms shortest paths and fastest paths/routes interchangeably

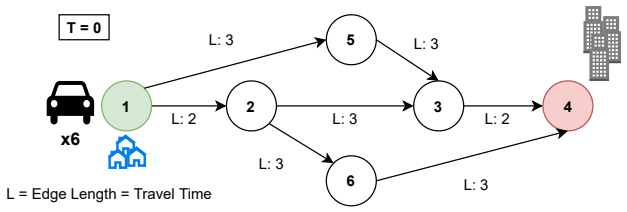


Figure 1: Motivating Example

and two vehicles along route 1-2-6-4. Although some travelers experience an additional time cost (equal to one unit) compared to the uncongested shortest path, these routes result in less congestion and delay for individuals and the system. We note that we consider multiple origins and destinations in the problem we study.

The problem of collectively processing shortest path queries (CP-SPQ) is formalized as minimizing the aggregate cost for concurrent queries in a network that is both time- *and* load-dependent. We develop a collective solution that enables proactive decisions to be made, based on the expected future loads, by adjusting edge values, and dynamically selecting the paths with the earliest expected arrival at their destination. To do so, we first define the notion of temporal load-aware networks (TLAN), which maintain expected future load along each edge within a given time interval, with a proof to satisfy the desirable first-in, first-out (FIFO) property. This property enables a TLAN to guarantee viable acyclic routes exists and that waiting on nodes cannot result in shorter routes. We utilize a context-aware arrival time function to penalize edges that are expected to be congested, and this controls transitions within the TLAN whilst satisfying FIFO requirements.

We then develop temporal load-aware adaptations of the widely used A* and top-k algorithms. These adaptations detect expected future congestion in the network, and dynamically provide alternative routes to avoid congestion by enabling queries, past and present, to interact via the TLAN. We finally propose a scalable solution for collectively and concurrently computing shortest paths that aims to optimize system-level travel times. As opposed to chronological processing of queries, our approach aims to find a more advantageous order for processing queries by iteratively finding paths that returns the lowest arrival time. The processing is embarrassingly parallel thus ensuring scalability. We discuss how our work can be integrated into navigational services, enabling them to benefit from collective processing. We enhance the efficiency of the collective solution by utilizing machine learning to restrict the search space of possible routing options. We maintain accuracy by continuously tracking computations between iterations to reduce redundancies.

Experiments on real-world data show that our collective approach achieves significant improvements in congestion reduction, utilization of the network's capacity, and distributing the load across a more diverse set of edges. For example, the proposed approach achieves time savings (compared to existing solutions) of up to 4.8 minutes per trip in Porto (17% saving), and 3.4 minutes in New York (20% saving); across all users this equates to saving thousands of hours for the city. The results are consistent when the collective system incorporates only a portion of the load (e.g., when an operator does not observe the entire traffic flow, or users do not follow recommended paths). Our solution is able to reach up to 97% load distribution, utilizing almost all capacity on the roads, which represents a 30% improvement on baseline algorithms.

Our contributions can be summarized as follows:

- We study the CP-SPQ problem with a novel formalization for temporal-load aware networks and a context-aware arrival time function, which enables the desirable FIFO property.
- We propose temporal load-aware adaptations of popular algorithms and a scalable solution that collectively processes shortest path queries with significant reductions in costs.
- We provide a framework for studying the CP-SPQ problem in large TLANs, including a set of performance metrics, covering total costs, network utilization, and fairness.

2 RELATED WORK

A wide variety of shortest path problems have been studied extensively with many efficient solutions [see 22, 33]. A particularly useful setting is to model the network with temporal data that brings time savings for the average journey [10]. For example, George and Shekhar [16] maintain a temporal network with a static structure and time-varying features (e.g., travel time), which improves space and algorithmic efficiency. In our work, the network also incorporates an arrival time function to capture time and load variations along edges, as well as the current and future state of the network.

Route diversification is a common approach for congestion alleviation in road networks [6]. Barth and Funke [2] present a routing algorithm for a network of autonomous vehicles, by generating a set of viable alternative routes and then assigning one for each query. This approach considers each individual vehicle (rather than as a collective optimization problem) and assigns the routes randomly. An alternative approach is to adapt Yen's top-k shortest paths algorithm [32] and select paths with high spatial diversity and low cost diversity [6]. This approach seeks to optimize a single query but does not address the collective routing of multiple queries for different origins/destinations. Nguyen et al. [25] adapt the A* algorithm to introduce diversity in route allocation. Their approach is to randomly perturb the edge costs to change the returned shortest path, but with minimal impact to the cost of the paths returned. Jeong et al. [20] propose a navigational system that necessitates physical infrastructure (e.g., through installation of "base stations") and with which seeks the path with "shortest" congestion contribution within an acceptable delay to the user. We propose and demonstrate the effectiveness of a collective approach, something that these works do not particularly address.

The principle of a user equilibrium [31] on a road network motivates other works. That is, if each user takes their best path, an equilibrium will be reached meaning no driver can improve their travel time by taking an alternative route. In practice, as the road network is dynamic and information on the state of the network is imperfect, this equilibrium is never reached. Dynamic user equilibrium approaches [13, 26] route and then re-route vehicles dynamically, responding in real-time to road conditions. These approaches, however, are not inherently collective as they optimize for each user rather than the overall system. Our approach also proactively avoids congestion by integrating temporal load-aware features of the network into the routing and shortest path computation.

The broad notion of system optimal routing has been explored in various branches of literature, such as the studies on "socially optimal" routing [7] and linear programming based optimization [1]. Guo et al. [17] use an iterative approach of routing and rerouting trajectories until an acceptable stopping criteria is reached. Our work is fundamentally different in that we collectively route vehicles based on the expected workload within the system in order to proactively avoid congestion. Their system also depends on two-way connectivity between the system and the users so that real-time traffic conditions are detected. Motallebi et al. [24] propose a solution that avoids intersecting routes and, to do so, they maintain a heatmap of normalized travel times over the network (i.e., "reservation" graph that tracks expected load and learned temporal patterns from historic data). However, shortest path computations do not dynamically account for these features; instead they calculate the lowest impact route from a candidate set. The approach also handles queries chronologically, as opposed to collectively.

3 TEMPORAL LOAD-AWARE SETTING

In this section, we introduce the temporal load-aware setting, and formally define the problem we study. The challenging task of formulating an environment that is both time- and load-dependent is essential in enabling our solutions to respond dynamically to the current and future state of the network. It is desirable that this dynamic environment is FIFO-compliant so that shortest path algorithms can find a realistic solution. We aim to achieve efficiency with a dynamic arrival time function that operates on a static network structure, which means that we do not need to maintain multiple impressions of the network and its features.

3.1 Problem Definition

To model a road network, we define a directed graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ with vertex set $\mathcal{V}$ representing intersections, and an attributed edge set $\mathcal{E}$ representing roads. Edge attributes include vehicle capacity F_{ij}, edge length δ_{ij}, speed limit z_{ij}, and minimum travel time Υ_{ij}. We use the $E = |\mathcal{E}|$ notation to denote set cardinality.

A hybrid discrete-continuous time domain is utilized to reduce the space complexity of the problem, and to introduce time intervals to better record load on edges (where more specific time predictions could not be guaranteed). Time intervals are given by $\mathcal{T} = \{\tau_1, \tau_2, \ldots, \tau_n, \ldots, \tau_T\}$, where the length of an interval is $\tau_{n+1} - \tau_n = I$. Edge loads are recorded for each discrete time interval, and the arrival time at a node is given in the continuous time domain, denoted as t. We define the load as the number of vehicles who occupy an edge at any point during the time interval.

The edge-load matrix (ELM), given by $\mathcal{L}(\mathcal{E}, \mathcal{T})$, contains the load for each edge $e_{ij} \in \mathcal{E}$, for each (future) time interval $\tau \in \mathcal{T}$. We use $l(e_{ij}, \tau)$ to denote the load on edge e_{ij} during τ. The ELM tracks the expected temporal variation in load across the network. It is also key in defining a temporal load-aware network – a fundamental part of the solution to the CP-SPQ problem.

Definition 3.1 (Temporal Load-Aware Network). A network given by $\mathcal{G}(\mathcal{V}, \mathcal{E})$ in which the load along each edge $e_{ij} \in \mathcal{E}$ is tracked with respect to each discrete time interval in $\mathcal{T}$ under the edge-load matrix $\mathcal{L}(\mathcal{E}, \mathcal{T})$.

The query set Q contains shortest path queries $q_{sdt} \in Q$ from v_s to v_d, starting at time t. For each query q_{sdt}, we determine a path p_{sdt}, which is assigned to the path set $\mathcal{P}$. Each path p_{sdt} consists of a sequence of edge-time interval pairs (e, τ) to connect v_s with v_d. Each query has a theoretical shortest path ϕ_{sdt} under free-flow conditions. We use $|\cdot|$ notation to denote the length of a path (synonymous with travel time). The assigned path may have a greater cost than ϕ_{sdt}, either due to congestion along the shortest path, or because an alternative path is selected as ϕ_{sdt} is too congested. Hence, the congestion penalty is:

$$\pi_{sdt} = |p_{sdt}| - |\phi_{sdt}| \tag{1}$$

An edge-level arrival time function, f_{ij}, controls transitions in the network. It is a user-specified, non-negative function of edge attributes and the current load. We denote the arrival time at v_i as a_i. Hence, $a_j = f_{ij}(a_i)$, and the arrival time of q_{sdt} at v_d is a_{sdt}.

Our aim is to compute a path p_{sdt} for each q_{sdt}, such that the total travel time of all $p_{sdt} \in \mathcal{P}$ is minimized, formally defined as:

$$\min \quad \sum_{k=1}^{Q} a_d^k \tag{2}$$

$$\sum_{j:e_{ij}\in\mathcal{E}} x_{ij}^k(\tau) - \sum_{j:e_{ji}\in\mathcal{E}} x_{ji}^k(\tau) = b_i^k \tag{3}$$

$$b_i^k = \begin{cases} 1 & \text{if } i = v_s^k \\ -1 & \text{if } i = v_d^k \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

$$x_{ij}^k(\tau) \cdot f_{ij}(a_i^k) = x_{ij}^k(\tau) \cdot a_j^k \tag{5}$$

$$x_{ij}^k(\tau) = \begin{cases} 1 & \text{if } a_i^k \le \tau \le a_j^k \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

$$l(e_{ij}, \tau) = \sum_{k=1}^{Q} x_{ij}^k(\tau) \tag{7}$$

where $x_{ij}^k(\tau)$ is a binary variable denoting whether e_{ij} is traversed by a vehicle k during τ. (3) and (4) ensure that a valid path from v_s^k to v_d^k is established. (5) ensures that, if an edge is traversed (i.e., $x_{ij}^k(\tau) = 1$), the arrival times satisfy the requirements of the arrival time function: $f_{ij}(a_i^k) = a_j^k$. (6) and (7) impose load constraints on edges. That is, if e_{ij} is traversed during τ, then $x_{ij}(\tau) = 1$. The total load $l(e_{ij}, \tau)$ is the number of vehicles using edge e_{ij} in τ, which is the sum of $x_{ij}^k(\tau) = 1$ over all Q vehicles.

The problem without the temporal load-aware constraint is NP-hard [19], which motivates our heuristic-based approach.

3.2 Arrival Time Function

A time- and load-dependent arrival time function dynamically computes travel times in TLANs, enabling proactive decisions about which edges to traverse. By using such an arrival time function we can maintain a static network structure, but vary weight edges dynamically which is space-efficient. Importantly, our solutions are not dependent on this specific arrival time function; any FIFO-compliant arrival time function is permissible.

Flow Regimes. When $l(e_{ij}, \tau) \le F_{ij}$, an edge is in a *free-flow regime*. In a road network, this means all vehicles are traversing the edge at (or below) the speed limit. When $l(e_{ij}, \tau) > F_{ij}$, an edge is in a *congested flow regime*. This means that any vehicles that subsequently join this edge will experience slower travel times (thus

ensuring safe distances between vehicles are maintained). Other methods for modeling the behavior of vehicles along edges would be appropriate as long as they are FIFO-compliant.

Delay Exponent. To control the rate at which speeds decrease along an edge, we introduce a load-dependent exponent, ϵ, that ensures travel times are penalized when edges are congested. Naturally, the more congestion that exists on an edge the greater the delay. In free-flow, $\epsilon = 1$; in congested flow, ϵ decreases as load increases:

$$\epsilon(\tau) = \begin{cases} 1 & l(e_{ij}, \tau) \leq F_{ij} \\ \frac{1}{l(e_{ij},\tau) - F_{ij}} & l(e_{ij}, \tau) > F_{ij} \end{cases} \tag{8}$$

Arrival Time Function. The arrival time function for an edge e_{ij} is:

$$f_{ij}(a_i) = \tau_i + (a_i - \tau_i)^{\epsilon(\tau_i)} + \Upsilon_{ij} = a_j \tag{9}$$

where, a_i and a_j are the arrival times in the continuous domain at vertices v_i and v_j respectively, and $\tau_i = \lfloor a_i \rfloor$ (i.e., the discrete time interval in which a_i lies).

In free-flow, the function returns the time in the continuous domain that a user arrives at v_i, plus the minimum travel time along an edge. As the load increases, the delay term tends to I. This bounds the delay term such that a user cannot wait until the following time interval to return an earlier arrival time – an important characteristic that is crucial to ensuring FIFO compliance.

Figure 2 shows how a_j varies under different conditions. As the edge load increases, the delay term increases and the time taken to traverse the edge increases non-linearly. As the load increases by a large amount, the effects of congestion on a_j start to plateau (e.g., traffic is likely to be traveling almost as slow as possible).

3.3 First-In, First-Out (FIFO) Compliance

FIFO networks have been integral for many routing tasks [8, 9, 27, 30]. Here, we demonstrate how our arrival time function for a TLAN is FIFO-compliant. FIFO compliance guarantees that: a) an acyclic shortest path exists in a TLAN, which guarantees a viable path exists for any (s, d, t) combination; and b) the sub-path of a shortest path is also a shortest path, which is necessary to guarantee results using Dijkstra's shortest path algorithm. Our formulation guarantees that all vehicles exit edges in the same order in which they enter (i.e., time gains along an edge cannot be attained by delaying the departure from a particular node).

Theorem 1. *The arrival time function, $f_{ij}(a_i)$, is FIFO-compliant.*

Proof. To show that $f_{ij}(a_i)$ is FIFO-compliant and waiting at a node cannot result in a quicker traversal time, we consider the two possible 'wait' cases.

Case 1 – Waiting within a time interval. If a user waits within a time interval until the next time interval, the only element of the arrival function that varies is the inner component of the delay term. This is a function of a_i, which increases as a_i increases. Hence, the arrival time can only increase as a_i increases within a time interval.

Case 2 – Waiting for the following time interval. If a vehicle waits at a node until the following time interval, then the delay exponent can vary. We need to examine whether it is possible for a vehicle arriving late within a congested time interval to wait until the following time interval (with less congestion) and be given an earlier arrival time. In this case, we are required to show that:

$$f_{ij}(a_1) \leq f_{ij}(a_2) \qquad \forall\ a_1 \in \tau_1,\ a_2 \in \tau_2 \tag{10}$$

where a_1 and a_2 are two different arrival times at v_i and $\tau_1 < \tau_2$. The limiting case is the one in which there is maximum congestion during τ_1, and no congestion during τ_2. For τ_1, as the exponent tends to 0, $f_{ij}(a_1) \rightarrow \tau_1 + I + \Upsilon_{ij}$. And, for a_2, when there is no congestion, the delay exponent equals 1 and so, $f_{ij}(a_2) = \tau_2 + \Upsilon_{ij}$. Knowing that $\tau_2 = \tau_1 + I$, we can show that $\max(f_{ij}(a_1)) \leq \min(f_{ij}(a_2))$. □

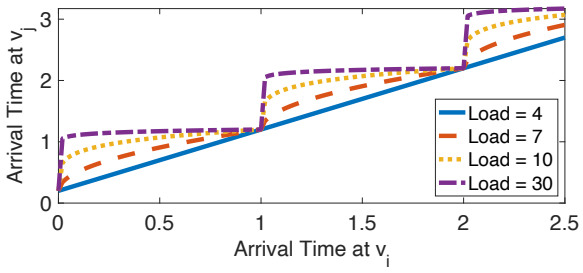


Figure 2: Effect on arrival time given varying loads along e_{ij}, with $F_{ij} = 5$, and $\Upsilon_{ij} = 0.3\tau$; arrival times given in terms of τ

Figure 2 also illustrates how f_{ij} ensures that waiting until the next timestep can not allow an earlier arrival time. Any vehicle leaving in the first timestep will always arrive earlier than vehicles traveling along an uncongested edge in the second timestep.

4 COLLECTIVELY PROCESSING SHORTEST PATHS QUERIES

In this section, we present our approach to address the CP-SPQ problem. Initially we address the shortest path query in a TLAN. The context-aware arrival time function enables the algorithms to dynamically account for the expected future load in the system, thus proactively avoiding congestion. The ordering of queries, however, is important as chronological ordering (or any other ordering) does not guarantee optimal results. To address this issue, we then present CS-MAT, our solution for collectively processing queries. CS-MAT searches for a path with the earliest arrival time from unallocated queries whilst avoiding re-computation of routes on-the-fly from a candidate set. Concurrently processing a large number of shortest path queries is a significant technical challenge, and so we demonstrate how our approach can be parallelized easily, which enables it to be applied to large networks and query workloads.

4.1 Shortest Path Query in a TLAN

In this section, we present algorithms that solve the shortest path query in TLANs. By operating in a TLAN and updating the ELM after each query is processed, these solutions are able to detect expected future load within the system. Congested edges are therefore proactively avoided as the arrival time function penalizes these edges. First, we present temporal load-aware A* (TLAA*) – an adaptation of the classic A* algorithm [18], and then we present a high-level description of Temporal Load-Aware Top-k (TLATk).

Algorithm 1 Temporal Load Aware A*

Input: $\mathcal{G}(\mathcal{V}, \mathcal{E})$, $\mathcal{L}(\mathcal{E}, \tau)$, q_{sdt}
Output: p_{sdt}
1: Initialize B, M, H
2: $H(s) \leftarrow \phi_{sdt}$
3: $\mathcal{V}' \leftarrow \mathcal{V}$
4: **while** $\mathcal{V}' \neq \emptyset$ **do**
5: $\bar{v} \leftarrow \arg\min_v(H)$ ▷ Shortest path in H
6: **if** $\bar{v} = d$ **then**
7: **return** p_{sdt}
8: **else**
9: $\mathcal{V}' \leftarrow \mathcal{V}' \setminus \bar{v}$
10: $\bar{a} \leftarrow q_t + B(\bar{v})$ ▷ Time arriving at current node
11: **for all** $v \in \mathcal{N}_{\mathcal{G}}(\bar{v})$ **do** ▷ Visit neighboring nodes
12: $a_j \leftarrow f_{ij}(\bar{a})$
13: **if** $a_j < B(\bar{v})$ **then** ▷ Quicker path found
14: Update B, M, H

4.1.1 Temporal Load-Aware A.* In TLAA* (Algorithm 1), new shortest paths are iteratively discovered and recorded with their actual cost (taking expected congestion into account), plus the expected cost to the destination node. At the next iteration, TLAA* visits the node with the lowest expected cost to the destination node. The length of the shortest free-flow path is used as the expected cost function. This is kept in a pre-computed pairwise matrix. As the network is FIFO-compliant, TLAA* guarantees that the shortest path for any query and any load distribution can be found.

Algorithm Outline. First, we initialize three tracking vectors: B, M, and H. B tracks the length of the best path to a given node in $\mathcal{G}(\mathcal{V}, \mathcal{E})$ and we set $B(v)$ to ∞ initially. M tracks the predecessor node along the currently known shortest paths and we initialize $M(v)$ as $\emptyset$. H gives the expected cost from a given node to v_d (Lines 1-2) and we initially set $H(v) = \infty$ for all $v \in \mathcal{V}$, except v_s for which we set $H(v_s) = \phi_{sdt}$ (Line 2). We first identify the shortest path in H (Line 5); in the first iteration, this will be the source node. If the identified node is v_d, the algorithm terminates (Lines 6-7). Otherwise, we remove the current node from $\mathcal{V}'$ and obtain the time it arrives at $\bar{v}$, denoted as $\bar{a}$ (Lines 9-10), given by query start time plus the length of the best known path to the current node. The load-aware arrival times at neighboring nodes to $\bar{v}$ are then computed ($\mathcal{N}_{\mathcal{G}}(\bar{v})$ denotes the neighborhood set). If a shorter path to any node is found, B, M, and H are updated (Lines 11-14).

Running Time Efficiency. The worst-case running time for TLAA* is $O(V \log V + ET)$. $V \log V$ gives the number of searches in the graph for a new shortest path when common speed-up techniques are used (e.g., min-heaps, adjacency lists [34]). Each search necessitates a query on the ELM to return the expected future load. The search on this matrix is bounded by T (i.e., the maximum number of time intervals) and, as no edge will be traversed more than once, this yields ET. Since the search is guided towards the target with an expected cost function, the running time is more efficient in practice. The memory required to implement TLAA* is concentrated on the ELM, and is upper-bounded at $O(ET)$. In practice, we find that operations on this matrix are executed in a matter of milliseconds (using our computational set-up described in Section 5.4).

4.1.2 Temporal Load-Aware Top-k. TLATk chooses the optimal path among each query's pre-computed top-k shortest paths given the known and expected load distribution. By pre-computing possible paths we speed up online processing, but lose some depth in the search compared to TLAA*. In a low congestion system where users are likely to take their free-flow best path anyway, this approach may yield useful results. As a pre-processing step, we compute the k-shortest paths under free-flow conditions (e.g., by using Yen's algorithm [32]) and store them in a matrix. As load builds up in the system, and some edges become congested, the fastest free-flow path will no longer necessarily be the best path to assign to the user. Within the temporal load-aware setting, we can calculate the expected cost of the k-shortest paths given the current (and expected) load, and assign to the user the fastest path. The running time complexity of this algorithm is $O(kV)$.

4.2 Collective Search For Minimal Arrival Time (CS-MAT)

TLAA* guarantees the shortest paths in a TLAN for any *individual* query. But, as the ELM is updated after processing each query, the load distribution can vary significantly depending on the order in which queries are processed. To address this issue, we propose CS-MAT, a collective algorithm that iteratively finds the lowest possible arrival time query from a candidate set.

From within a candidate set of queries $Q' \subseteq Q$, CS-MAT uses any temporal load-aware algorithm (e.g., TLAA*) to find the path with the earliest arrival time at its destination node and assign this path. In most cases, this is not the query that is encountered chronologically, which often leads to the same query being 're-processed' multiple times. Once a query has been assigned, it is added to the ELM ensuring future queries can dynamically take account of its load impact on the system.

A naïve way to perform these steps would be to simply re-process all unallocated queries after a query has been assigned a path and the ELM has been updated. This is inefficient and leads to many redundant calculations. Hence, CS-MAT restricts the search in Q for Q'.

Predicting Congestion Penalty. To restrict the search in Q for candidate queries, queries whose departure time is after the predicted arrival time of the current query are ignored, as they are unlikely to give an earlier arrival time. We can utilize a machine learning model, trained on representative queries, to predict a query's congestion penalty, π^{pred}. The training query set is processed using TLAA*, and the congestion penalties are captured. Using a feed-forward neural network, we convert v_s and v_d into a sparse matrix using one-hot encoding, and append time, t, as a stand-alone variable to create the feature vector. π^{pred} is the target variable and, using this model, a query's estimated arrival at v_d is:

$$\xi_{sdt} = a_s + |\phi_{sdt}| + \pi^{pred}_{sdt} \tag{11}$$

Minimizing Redundant Computations. Once a path has been assigned, only paths that have edge-time locations in common with the assigned path (i.e., where ELM values change) will experience changes in arrival time. Hence, CS-MAT does not re-compute all paths in Q'. It is sufficient to only re-process paths that intersect with the previously assigned path.

Rolling Query Batches. CS-MAT can also be performed by using rolling batches within Q. A pre-specified time parameter, y, can control how 'far' into Q to search for candidate queries (measured from the departure time of the current query). This restricts the size of Q', which may be useful in low CPU environments.

4.2.1 Design Details. We now proceed to outline CS-MAT, as described in Algorithm 2.

Batch Forming. After initializations (Lines 1-2), CS-MAT selects the next chronological query in Q from which to define Q_{batch}. The algorithm forms Q_{batch} from the queries entering the system in a time window, controlled by y. The unprocessed queries in Q_{batch} are sorted by their pre-computed free-flow arrival time (Line 5) and we then iterate through Q_{batch} (from Line 7).

Simple Case. The goal at each iteration is to find the unprocessed query in Q_{batch} that returns the earliest arrival time, given the expected future load. In the simplest case, the first query (Line 15) is tested to see if any edges are in a congested regime (Lines 11-14). If no edges are, we know this query must have the earliest arrival time in Q_{batch} (as Q_{batch} is already sorted by free-flow arrival time) and so we assign its path, update the ELM, remove the query from Q_{batch}, and move to the next query (Lines 15-18).

Restricting the Search Space. If the first query has any congested edges, we need to define Q', which is the set of candidate queries from which to find the earliest arrival time. There are two sub-cases (controlled by *reCheck*). In sub-case 1 (Lines 20-22), we encounter a 'base' query for the first time and need to find all possible candidate queries. To determine Q', the expected arrival time is calculated, ξ_{sdt} of the current q_{sdt}. To populate Q', all queries in Q_{batch} with an earlier free-flow arrival time than the expected arrival time are selected. In sub-case 2 (Lines 23-24), Q' is already known (e.g., the same base query is being re-processed) and the queries therein already calculated. We therefore only want to identify those queries in Q' that need to be recalculated. The calculated paths in Q' are recorded in a path matrix $P(e,\tau)$, where path ID is recorded with its (e,τ) tuple. Next, Q' is updated by redefining it as the set of all the queries whose ID intersects with previously assigned path ID. That is, all $q' \in Q'$ for which at least one edge in p' intersects, both spatially and temporally, with the previously assigned path.

Determining Temporal Load-Aware Shortest Paths. Once Q' has been defined, the temporal load-aware paths for each q' is determined using TLAA* and added to $P(e,\tau)$ (Lines 27-28). Each path can be determined in parallel without any loss of information. The returned paths are compiled and the path with the earliest arrival time is assigned (Lines 29-31). Finally, we update $\mathcal{P}$ with this best path, update $\mathcal{L}(\mathcal{E},\tau)$, and remove the corresponding query from both Q and Q_{batch} (Lines 32-35).

4.2.2 Scalability. CS-MAT enables query processing to be embarrassingly parallel. When the shortest path from one query in Q' is determined, it does not affect the processing of another. We can achieve this while still being collective because the shortest path calculations within a single iteration are only dependent on the ELM, rather than one another. Once a path has been assigned its impact on expected system load, and on other queries, in the candidate set Q' is accounted for in subsequent iterations. After batching, each iteration has two operations. The 'outer operation' selects the candidate query set, prepares it for parallelization, and then recompiles the shortest path results to identify which query in Q' to assign. The 'inner operation' calculates the shortest paths and can use any shortest path algorithm (e.g., TLAA*). The complexity is upper-bounded by $O(Q(V \log V + ET))$ due to the maximum number of outer iterations (in practice far fewer than Q). The inner cost is the complexity of TLAA*. We observe that the runtime of the outer operation is typically in the order of milliseconds, and always less than half a second in our experimental setting. Hence, when fully parallelized, the actual algorithm runtime is not much more than that of a stand-alone shortest path calculation.

Algorithm 2 Collective Search For Minimal Arrival Time (CS-MAT)

Input: $\mathcal{G}(\mathcal{V},\mathcal{E})$, Q, y
Output: $\mathcal{P}$

1: Initialize $\mathcal{L}(\mathcal{E},\tau)$
2: $\mathcal{P} = \emptyset$
3: **while** $Q \neq \emptyset$ **do**
4: Define Q_{batch} ▷ Determined using y
5: Sort Q by free-flow arrival time
6: $reCheck \leftarrow False$
7: **while** $Q_{batch} \neq \emptyset$ **do**
8: **if** $reCheck = False$ **then**
9: $q_{sdt} \leftarrow$ Next query in Q_{batch}
10: $congPath \leftarrow False$
11: **for all** $e_{i,j} \in \phi_{sdt}$ **do**
12: **if** $l(e,\tau) \geq F_{ij}$ **then**
13: $congPath \leftarrow True$
14: **break**
15: **if** $congPath = False$ **then** ▷ Free-flow best path available
16: $\mathcal{P} \leftarrow \mathcal{P} \cup \phi_{sdt}$
17: Update $\mathcal{L}(\mathcal{E},\tau)$
18: $Q_{batch} \leftarrow Q_{batch} \setminus q_{sdt}$
19: **else** ▷ Best path congested
20: **if** $reCheck = False$ **then**
21: Initialize $P(e,\tau)$ ▷ Path-Edge Matrix
22: Define Q' ▷ (Sub-case 1)
23: **else**
24: Define Q' ▷ (Sub-case 2)
25: $p^* \leftarrow \emptyset$; $q^* \leftarrow \emptyset$; $a^*_{sdt} \leftarrow \infty$
26: **for all** $q' \in Q'$ **do**
27: $p' \leftarrow$ TLAA*$(\mathcal{G}(\mathcal{V},\mathcal{E}), \mathcal{L}(\mathcal{E},\tau), q')$
28: Add p' to $P(e,\tau)$
29: Calculate a'_{sdt}
30: **if** $a'_{sdt} < a^*_{sdt}$ **then**
31: $p^* \leftarrow p'$; $q^* \leftarrow q'$; $a^*_{sdt} \leftarrow a'_{sdt}$
32: $\mathcal{P} \leftarrow \mathcal{P} \cup p^*$
33: Update $\mathcal{L}(\mathcal{E},\tau)$
34: $Q \leftarrow Q \setminus q'$
35: $Q_{batch} \leftarrow Q_{batch} \setminus q'$
36: Update $reCheck$
37: **return** $\mathcal{P}$

5 EXPERIMENTS

This section outlines our experimental setting, the datasets, performance metrics, and baselines. The experiments use representative

query workloads as the basis for travel demand, which mirrors common practice in real-life traffic management in which OD matrices are used as the basis for demand information.

5.1 Modeling Road Networks

We integrate a number of concepts that are prevalent in the traffic management community, including traffic flow regimes [23, 29], safe headway [11], and transition penalties [4]. The base headway, η, is the number of seconds required between vehicles for them to safely traverse an edge. A transition penalty, ψ, is incurred when moving from one road to another, and this captures elements of travel such as waiting at traffic lights or slowing to turn a corner. The free-flow capacity, F_{ij}, is the maximum number of vehicles that can safely traverse an edge at the speed limit in a given time interval. We model this as the number of vehicles that can exist on the edge while maintaining safe headway distance, plus the number of vehicles that can safely join the edge in the given time interval τ. Formally, it is given as:

$$F_{ij} = \frac{\delta_{ij}}{z_{ij} \times \eta} + \frac{I}{\eta + \psi} \tag{12}$$

Assumptions. To establish a generalizable basis for collective routing, we assume all queries result in a journey, all users begin their journeys as soon as their path is given to them, and all users follow the assigned path. FIFO compliance means there is no overtaking in the network. In determining F_{ij}, we assume that all vehicles are of equal size. This assumption can be relaxed to account for other vehicle types (e.g., trucks, buses) by following best practice methods from traffic modelling literature.

5.2 Data

We use real taxi trip data from two cities: Porto, Portugal [15] and New York, USA [28]. The cities have contrasting topologies: New York has an orderly grid-based road network, whereas Porto has a much less orderly network.

We extract 3km × 3km regions from OpenStreetMap (OSM) using `osmnx` [3]. These regions are centered on busy sites that show temporal variation: Penn Station for New York, and São Bento Station for Porto. Selecting these regions enable us to cover most of Porto's city center, and a significant proportion of Manhattan in New York. We apply geographical and temporal filters: we only select trips that start and end within these regions, and that start in the range 7-11am on a weekday (i.e., to simulate the rush-hour). We assume all journeys take place on the same day. We take edge attributes from OSM (and make inferences where data is unavailable).

As Porto is a smaller city than New York, we use smaller query sets for Porto (2k, 5k, 10k, and 25k), and larger query sets in New York (10k, 25k, 50k, and 100k) to represent their real-world congestion levels. For example, millions of journeys occur daily across New York, so 100k trips in a 9km^2 region across a four-hour window can be broadly representative of very high congestion.

5.3 Evaluation Measures

We evaluate the methods using the following measures. Average journey time (AJT), in minutes, is defined as:

$$AJT = \frac{1}{P} \sum_{p \in \mathcal{P}} |p_{sdt}| \tag{13}$$

Free-flow capacity utilization (FFCU) assesses what proportion of the free-flow capacity in the network has been utilized. To define FFCU, we first define $\sigma(e_{ij}, \tau)$, which represents the proportion of free-flow capacity that is utilized along an individual edge:

$$\sigma(e_{ij}, \tau) = \begin{cases} \frac{l(e_{ij},\tau)}{F_{ij}} & \text{if } l(e_{ij}, \tau) \leq F_{ij} \\ 1 & \text{if } l(e_{ij}, \tau) > F_{ij} \end{cases} \tag{14}$$

FFCU is then defined as:

$$FFCU = \frac{1}{ET} \sum_{\tau \in \mathcal{T}} \sum_{e_{ij} \in \mathcal{E}} \sigma(e_{ij}, \tau) \tag{15}$$

Load distribution (LD) returns the proportion of edges that are being utilized (i.e., have a non-zero load), defined as:

$$LD = \frac{1}{ET} \sum_{\tau \in \mathcal{T}} \sum_{e_{ij} \in \mathcal{E}} \rho(e_{ij}, \tau) \tag{16}$$

where $\rho(e_{ij}, \tau) = 1$ if $l(e_{ij}, \tau)$ is non-zero, and zero otherwise.

We evaluate the fairness of a given solution by assessing the distribution of each user's congestion penalty.

5.4 Experimental Set Up

In our experiments, we evaluate TLATk, TLAA*, and CS-MAT. To process queries, we sort Q in time-ascending order (i.e., those queries that start earliest are processed first). CS-MAT does not require the query set to be (time-)sorted as it is a collective solution.

The congestion alleviation methods used by Google Maps, TomTom, etc. are proprietary. However, we model what we understand from the traditional practice which is to route vehicles based on the currently known load distribution when the query is asked [20], but without explicit consideration for the future impact of that routing. We call this method Static Load-Aware Dijkstra (SLAD) and it forms one of our baselines. As TLAA* calculates transitions through the network using a load-aware arrival time function, so does SLAD except the underlying ELM reflects only the static load distribution at the timestep in which the query is initiated. It can therefore only adapt to congestion known at the point at which the query is asked. We also implement Free-Flow Naïve Dijkstra (FFND) as another baseline, which assigns each path its free-flow best path given by the basic version of Dijkstra's shortest path algorithm [12]. In assessing both of these baselines, we calculate the actual path cost using the actual load within the system and the arrival time function. We do not compare the methods presented in [24] as these depend on historical road network data, which is unavailable to us, or [17] as their optimization approach is fundamentally different.

Finally we set: $k = 5$, $I = 360$s, $\eta = 3$s [14], and $\psi = 0.5\Upsilon_{ij}$ seconds [21]. Within CS-MAT, y is set to 14,400s (=4 hours), which means all queries are ingested as a single batch. The algorithms are implemented in Python on a Linux machine with 64GB RAM and six CPU cores (all of which are used in our experiments).

6 RESULTS

We first analyze the performance of our temporal load-aware shortest path algorithms, then we evaluate the effect of computing shortest paths collectively using CS-MAT. We also investigate scenarios in which only a portion of the vehicles are routed collectively. Our analysis initially focuses on average journey times, and we then consider other aspects, such as network utilization and fairness.

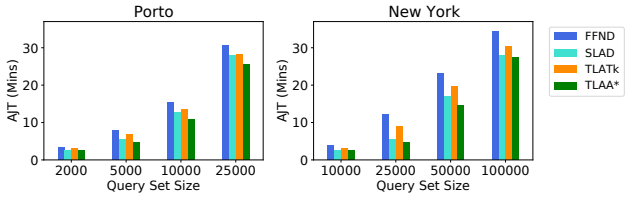


Figure 3: AJT of temporal load-aware algorithms

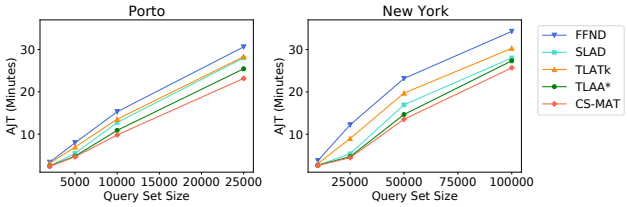


Figure 4: AJT across varying query set sizes

6.1 Temporal Load-Aware Algorithms

Temporal load-aware adaptations of popular algorithms are the 'engine' of any collective query processing, including our solution – CS-MAT. They can also be used as stand-alone solutions within existing navigational systems by modeling environments as TLANs.

Figure 3 shows that both TLATk and TLAA* report a lower AJT than FFND, and that TLAA* outperforms SLAD. Overall, TLAA* is consistently the best performing algorithm for computing the shortest paths in TLANs. We can evaluate the impact of operating in a TLAN by comparing SLAD and TLAA*. It is consistently beneficial to consider expected future traffic load, and the benefit tends to increase in more congested systems. However, under very high congestion, the difference in performance starts to plateau (e.g., in New York, there is only a 2.5% improvement under very high congestion, as opposed to 13.6% in high congestion regimes). This is expected as it is naturally harder to find free-flow capacity, or even viable alternative routes, when a significant number of edges are already heavily congested. Actual time savings are also strong: in New York, TLAA* saves 8.5 and 6.9 minutes (37% and 20%) compared to FFND when congestion is high and very high respectively, and 2.3 and 0.7 minutes compared to SLAD (15.7% and 2.5%).

6.2 Collective Query Processing

Comparing CS-MAT with other solutions indicates the impacts of collectively routing. Figure 4 shows that CS-MAT is the most effective algorithm at reducing AJTs in all scenarios. In Porto, CS-MAT improves upon TLAA* by 11.2% and 9.8% in high and very high congested systems, respectively. In New York, CS-MAT is 8.3% and 6.5% better than TLAA* in the same congestion regimes. As congestion increases, the benefit of collective processing becomes clearer, which is to be expected as more edges are likely to be over-capacitated and more re-routing is necessary.

We also compare the performance of CS-MAT with the baselines. Compared to FFND, CS-MAT significantly improves AJT; in Porto, CS-MAT creates AJT savings of up to 7.4 minutes in systems with high congestion, and we also see significant savings in medium congestion (42%, giving a saving of 5.5 minutes). In New York, CS-MAT makes bigger savings compared to FFND, where AJT is reduced by up to 63.5% (medium congestion). Against SLAD, CS-MAT is saving 2.9 and 4.9 minutes in high and very high congestion systems in Porto (23.1% and 17.3%), and 3.4 and 2.4 minutes in New York (20.2% and 8.4%). These time savings aggregate significantly at the system level (e.g., in New York, an AJT saving of three minutes for each of the 50k users gives a system-level saving of 2,500 hours).

In less-congested scenarios, the benefit of using CS-MAT over TLAA* is less pronounced. In a low congestion system, any algorithm is likely to return the free-flow best path for a query, as such ordering of queries matters less as the distribution of load in the system is less likely to adversely impact travel times for other routes. Where CPU capacity is limited, TLATk may be considered as it performs well against ND as it is still able to make some dynamic decisions even if these are limited to the k pre-computed paths.

6.3 Uncontrolled Load

While modeling and managing travel demand are attracting attention in modern traffic management, there are still limitations today in terms of monitoring the traffic loads. As autonomous/assisted driving is not yet widespread, we cannot expect every user to have their route assigned according to the same collective goal. Hence, we evaluate our algorithms under lower levels of load control. We generate a set of representative queries and process these using TLAA*. Where we reduce our level of control, we calculate a 'base' load in the system based off of these representative ELMs. The loads are reduced by γ%, where γ is a control factor. For example, if an edge has a predicted load of 10, the uncontrolled load would be 7 for 30% control, 5 for 50% control, etc.

Figure 5 shows that CS-MAT remains the best performing algorithm in terms of AJT. While the performance difference between CS-MAT and TLAA*/FFND tends to reduce at lower levels of control, the difference compared to SLAD actually increases with reduced levels of control which is an important finding. Not only is collective and dynamic routing effective even at lower levels of traffic control, but by not doing so when other operators are, existing approaches can actually return even worse results at lower levels of control. This is because the traffic load experienced by other operators is no longer as predictable as it once was, due to the TLA systems that dynamically respond to and re-route traffic.

6.4 Network Utilization

Recent studies [5] indicate that utilizing spare capacity can be an inexpensive way of reducing congestion and positively impacting quality of life and air pollution. Accordingly, we consider how collective routing affects network utilization and load distribution.

Figure 6a shows that CS-MAT and TLAA* are able to find uncongested edges better than the baselines. We expect this as these algorithms are making dynamic edge-level decisions driven by the arrival time function, which penalizes congested edges. The difference in performance compared to FFND and TLATk increases with mid-to-high levels of congestion, before plateauing when congestion intensifies. FFCU improves by 42% and 30% compared to FFND

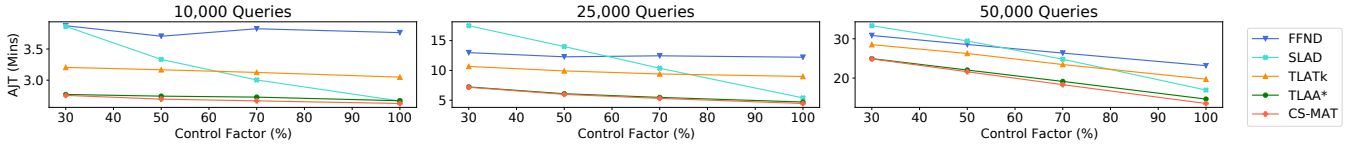
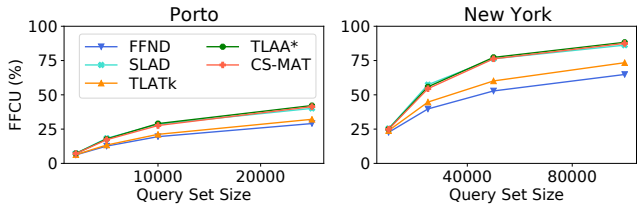
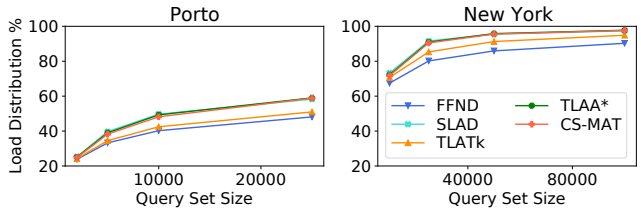


Figure 5: Variation in AJT in New York as the control factor varies; same legend in all sub-figures



(a) Free Flow Capacity Utilization



(b) Load Distribution

Figure 6: Network utilization for Porto and New York; same legend in all sub-figures

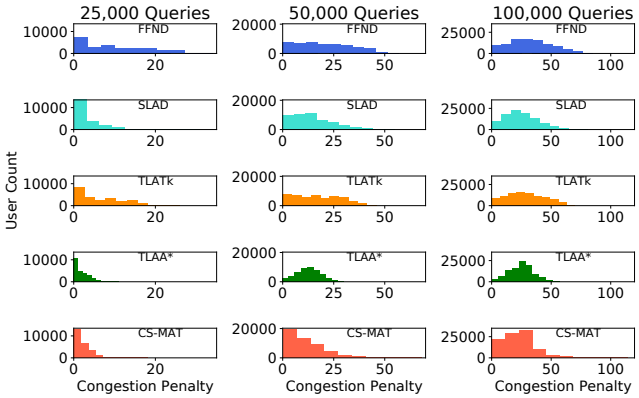


Figure 7: Distribution of congestion penalties (New York)

and TLATk in Porto in a highly congestion system (44% and 27% for New York). There is little difference between CS-MAT and TLAA*; TLAA* is slightly better at finding capacity, despite generally reporting slower journey times. This indicates there exists a trade-off between finding capacity and reducing AJT. That is, at some point, it is no longer beneficial to find free-flow capacity for its own sake.

Figure 6b shows that TLAA*, CS-MAT, and SLAD distribute load across a more diverse set of edges than TLATk and FFND, and the difference in performance increases in mid-to-high congestion level systems, but plateaus somewhat with very high congestion. In New York, although query set sizes are high, it is notable that CS-MAT is able to reach up to 97% load distribution, which is far higher than in Porto. The performance improvements achieved by our algorithms in New York compared to Porto emphasizes the importance of topology. While this may highlight the benefits of planned cities with organized, uniform road networks, it also indicates further research potential for algorithms that are specifically designed to find capacity within less-ordered networks.

6.5 Fairness

Inherent to any collective solution is that some individuals will be assigned paths that are not their theoretical free-flow shortest path. While we attempt to optimize journey times at the system level, it is important to understand how fair each system is for individuals.

Figure 7 shows the distribution of the congestion penalties (in minutes). Due to space limitations, we only show results for New York; Porto exhibits similar patterns. When congestion is high, CS-MAT has a wider spread of penalties while showing a positive skew, whereas TLAA* has a smaller spread and we observe a tendency towards a Gaussian distribution, particularly in higher congestion systems. Comparing TLAA* to CS-MAT under high congestion, the mean penalty for CS-MAT is lower (TLAA*: 12.2, CS-MAT: 11.1), but TLAA* has lower standard deviation (TLAA*: 5.9, CS-MAT: 9.6). TLAA* tends to offer a fairer distribution as fewer individuals receive 'harsh' penalties and, with a lower standard deviation, most users can expect similar penalties to others in the system. This indicates a trade-off may exist between reducing AJT and ensuring fairness in the system; this warrants further study.

6.6 Runtime Analysis

We also analyze algorithm runtime within the context of AJT improvements. For this, we measure the 'end-to-end' time, which is the time taken for the algorithm to return a route, and for the journey to be completed. We parallelized the execution across six nodes, which is typically fewer than would be available in a real-world deployment of any solution.

Formally, the end-to-end time is the sum of the 'per query' algorithm runtime (e.g., processing time) and the AJT. We take the average over the query set, and present these values in Table 1. CS-MAT is the best-performing algorithm for all levels of congestion, although TLAA* performs well. These results demonstrate the potential benefit of collective processing in providing socially conscious routing. With sufficient computing power and optimized implementations of the underlying shortest path algorithm, the collective approach can be more effective and significant additional time savings are feasible.

Table 1: End-to-end time, in minutes (New York)

No. Queries	ND	SLAD	TLATk	TLAA*	CS-MAT
10,000	3.76	2.67	3.05	2.67	**2.63**
25,000	12.21	5.40	8.99	4.70	**4.50**
50,000	23.16	16.96	19.72	14.66	**13.95**
100,000	34.28	28.04	30.30	27.34	**26.12**

7 CONCLUSION

In this paper, we addressed the problem of collectively processing shortest path queries with the aim of minimizing system-wide congestion. To solve the CP-SPQ problem, we formalized the temporal load-aware setting, in which the expected future load in the network is tracked, and an edge-level arrival time function penalizes congested edges. When queries are answered, and paths assigned, the network's tracking matrices are updated and future queries can take account of this expected load within the system.

We presented adaptations of classic algorithms to operate in TLANs. TLAA* adapts A*, enabling it to utilize future traffic states in order to dynamically and proactively avoid congestion thus finding alternative routes. TLATk pre-computes the k best free-flow paths, dynamically calculates the cost of these paths given current load in the system, and selects the best path for the query. Our solutions are more effective (i.e., they return a lower AJT) and can be as efficient as the baselines if fully parallelized. Our work is not just for the future; it would be beneficial for any navigational service to model their environment in a temporal load-aware way, and adapt their existing algorithms to be temporal- and load-aware. Experiments show that this can be effective even when operators have access to only a proportion of the traffic flow.

Finally, we introduced CS-MAT, which considers queries in a collective manner, by iteratively searching for the lowest arrival time from a query batch. We used machine learning and designed data types to track processing results between iterations in order to minimize redundant calculations, and discussed how CS-MAT is embarrassingly parallel, which indicates its potential for large-scale deployment. Not only does CS-MAT reduce AJTs for individuals by up to 20% in New York and 23% in Porto compared to non-collective solutions, it also better utilize the available capacity in the network finding up to 44% more free-flow capacity. These results emphasize the potential benefit that collectively processing shortest path queries on road networks can have on urban life.

ACKNOWLEDGMENTS

This work is supported in part by the UK Engineering and Physical Sciences Research Council under Grant No. EP/L016400/1. We also thank Elif Eser for her work in the early stages of the project.

REFERENCES

[1] Enrico Angelelli, Valentina Morandi, Martin Savelsbergh, and Grazia Speranza. 2021. System optimal routing of traffic flows with user constraints using linear programming. *European Journal of Operational Research* 293, 3 (2021), 863–879.
[2] Florian Barth and Stefan Funke. 2019. Alternative Routes for Next Generation Traffic Shaping. In *ACM SIGSPATIAL*. 1–8.
[3] Geoff Boeing. 2017. OSMnx: New methods for acquiring, constructing, analyzing, and visualizing complex street networks. *Computers, Environment and Urban Systems* 65 (2017), 126–139.
[4] Mark Brackstone and Mike McDonald. 1999. Car-following: a historical review. *Transportation Research Part F: Traffic Psychology and Behaviour* 2, 4 (1999), 181–196.
[5] Luboš Buzna and Rui Carvalho. 2017. Controlling congestion on complex networks: fairness, efficiency and network structure. *Scientific reports* 7, 1 (2017), 1–15.
[6] Dan Cheng, Olga Gkountouna, Andreas Züfle, Dieter Pfoser, and Carola Wenk. 2019. Shortest-Path Diversification through Network Penalization: A Washington DC Area Case Study. In *ACM SIGSPATIAL IWCTS*. 1–10.
[7] Serdar Çolak, Antonio Lima, and Marta C González. 2016. Understanding congested travel in urban areas. *Nature* 7, 1 (2016), 1–8.
[8] Brian C Dean. 1999. *Continuous-time dynamics shortest path algorithms*. Ph.D. Dissertation. Massachusetts Institute of Technology.
[9] Brian C Dean. 2004. Shortest paths in FIFO time-dependent networks: Theory and algorithms. *Rapport Technique* (2004), 13.
[10] Ugur Demiryurek, Farnoush Banaei-Kashani, and Cyrus Shahabi. 2010. A case for time-dependent shortest path computation in spatial networks. In *ACM SIGSPATIAL*. 474–477.
[11] Department for Transport. [n.d.]. The Highway Code. https://www.gov.uk/guidance/the-highway-code/
[12] Edsger W Dijkstra et al. 1959. A note on two problems in connexion with graphs. *Numerische mathematik* 1, 1 (1959), 269–271.
[13] Chaimae El Hatri and Jaouad Boumhidi. 2017. Traffic management model for vehicle re-routing and traffic light control based on Multi-Objective Particle Swarm Optimization. *Intelligent Decision Technologies* 11, 2 (2017), 199–208.
[14] Stephen H Fairclough, Andrew J May, and C Carter. 1997. The effect of time headway feedback on following behaviour. *Accident Analysis & Prevention* 29, 3 (1997), 387–397.
[15] Geolink. 2015. *ECML/PKDD 15: Taxi Trajectory Prediction*. Retrieved August 15, 2019 from http://www.geolink.pt/ecmlpkdd2015-challenge/dataset.html
[16] Betsy George and Shashi Shekhar. 2008. *Time-Aggregated Graphs for Modeling Spatio-Temporal Networks*. 191–212.
[17] Chang Guo, Demin Li, Guanglin Zhang, Xiaoning Ding, Reza Curtmola, and Cristian Borcea. 2020. Dynamic Interior Point Method for Vehicular Traffic Optimization. *IEEE Transactions on Vehicular Technology* 69, 5 (2020), 4855–4868.
[18] Peter E Hart, Nils J Nilsson, and Bertram Raphael. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics* 4, 2 (1968), 100–107.
[19] Olaf Jahn, Rolf H Möhring, Andreas S Schulz, and Nicolás E Stier-Moses. 2005. System-optimal routing of traffic flows with user constraints in networks with congestion. *Operations Research* 53, 4 (2005), 600–616.
[20] Jaehoon Jeong, Hohyeon Jeong, Eunseok Lee, Tae Oh, and David HC Du. 2015. SAINT: Self-adaptive interactive navigation tool for cloud-based vehicular traffic optimization. *IEEE Transactions on Vehicular Technology* 65, 6 (2015), 4053–4067.
[21] David Levinson. 2018. How much time is spent at traffic signals? https://transportist.org/2018/03/06/how-much-time-is-spent-at-traffic-signals/
[22] Amgad Madkour, Walid G Aref, Faizan Ur Rehman, Mohamed Abdur Rahman, and Saleh Basalamah. 2017. A survey of shortest-path algorithms. (2017). arXiv:1705.02044
[23] Adolf Darlington May. 1990. *Traffic flow fundamentals*. Prentice Hall, Chapter 10.
[24] Sadegh Motallebi, Hairuo Xie, Egemen Tanin, and Kotagiri Ramamohanarao. 2020. Streaming route assignment with prior temporal traffic data. In *ACM SIGSPATIAL IWCTS*. 1–10.
[25] Uyen TV Nguyen, Shanika Karunasekera, Lars Kulik, Egemen Tanin, Rui Zhang, Haolan Zhang, Hairuo Xie, and Kotagiri Ramamohanarao. 2015. A randomized path routing algorithm for decentralized route allocation in transportation networks. In *ACM SIGSPATIAL IWCTS*. 15–20.
[26] Juan Pan, Mohammad A Khan, Iulian Sandu Popa, Karine Zeitouni, and Cristian Borcea. 2012. Proactive vehicle re-routing strategies for congestion avoidance. In *IEEE Conference on Distributed Computing in Sensor Systems*. 265–272.
[27] R Campi Sperb. 2010. *Solving time-dependent shortest path problems in a database context*. University of Twente.
[28] New York City TLC. 2015. *TLC Trip Record Data*. Retrieved January 15, 2019 from https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page
[29] Haizhong Wang, Jia Li, Qian-Yong Chen, and Daiheng Ni. 2009. Speed-Density Relationship: From Deterministic to Stochastic. In *TRB Annual Meeting*.
[30] Yong Wang, Guoliang Li, and Nan Tang. 2019. Querying shortest paths on time dependent road networks. *PVLDB* 12, 11 (2019), 1249–1261.
[31] John Glen Wardrop. 1952. Some theoretical aspects of road traffic research. *Proceedings of the Institution of Civil Engineers* 1, 3 (1952), 325–362.
[32] Jin Y Yen. 1971. Finding the k shortest loopless paths in a network. *Management Science* 17, 11 (1971), 712–716.
[33] F Benjamin Zhan and Charles E Noon. 1998. Shortest path algorithms: an evaluation using real road networks. *Transportation Science* 32, 1 (1998), 65–73.
[34] Wei Zhang, Chong Jiang, and Yunxiang Ma. 2012. An improved Dijkstra algorithm based on pairing heap. In *International Symposium on Computational Intelligence and Design*, Vol. 2. 419–422.